Entwurf und Implementierung eingebetteter Systeme*

Jan Philipps Alexander Schmidt

Institut für Informatik
Technische Universität München, 80290 München

{philipps,schmiale}@informatik.tu-muenchen.de

1 Einleitung

Ein Großteil der heute eingesetzten Computer befindet sich nicht auf Büroschreibtischen oder in Rechenzentren, sondern im inneren von Videorekordern, Autos, Mikrowellenherden und anderen elektromechanischen Systemen. Für derartige Systeme ist die korrekte Funktion des Softwareanteils besonders wichtig. Selbst wenn nicht unmittelbar menschliches Leben gefährdet ist, führen Softwarefehler leicht zu Zeitverlusten bei der Produkteinführung oder gar zu teuren Rückrufaktionen.

Der Entwurf solcher eingebetteter Systeme erfordert Techniken sowohl der Ingenieurwissenschaften als auch der Informatik. Wünschenswert ist dabei ein Entwurfsprozeß, der etablierte graphische Beschreibungstechniken wie Zustandsmaschinen oder Blockdiagramme mit formalen und mathematisch fundierten Modellen der Informatik verbindet. Durch diese Verbindung erhalten die suggestiven graphischen Darstellungen eines Systems eine präzise Bedeutung, die Mißverständnisse zwischen den Entwicklern vermeiden hilft, aber auch Grundlage für Werkzeugunterstützung bis hin zur Code-Erzeugung ist. Ein weiterer Vorteil einer mathematisch präzisen Semantik von Beschreibungstechniken ist die formale Beweisbarkeit von Systemeigenschaften.

Am „Tag der Informatik“, einem Tag der offenen Tür im November 1997 an der TU München bot sich die Gelegenheit, an einem kleinen, aber repräsentativen Beispiel die Vision des durchgängigen, verifizierbaren und weitgehend werkzeuggestützten Entwurfs und der Code-Erzeugung eingebetteter Systeme am Modell zu demonstrieren. Als Beispiel diente ein Aufzug, da dessen Benutzerschnittstelle allgemein bekannt ist. Die Gäste konnten während der Veranstaltung das Aufzugmodell bedienen und – allerdings vergeblich – versuchen, Fehler in der Steuersoftware zu finden.

* Diese Arbeit wurde vom BMBF im Rahmen des Verbundprojektes KorSys unterstützt.

Auf den folgenden Seiten beschreiben wir die verwendeten Beschreibungstechniken (§2), die Modellierung des Aufzugs (§3) und die Code-Erzeugung aus dem Modell (§4). In §5 gehen wir auf die Verwendung von Verifikationswerkzeugen ein. Schließlich erläutern wir in §6, wie die verbleibenden Lücken bei der Automatisierung des Entwurfsprozesses geschlossen werden können.

2 Formale Beschreibungstechnik

Ausgangspunkt der formalen Entwicklung gemäß der FOCUS-Methodik [7] sind über Nachrichtenkanäle kommunizierende Komponenten, die aus einer Zerlegung des Gesamtsystems in funktional abgeschlossene und überschaubare Einheiten hervorgehen.

Die Kanäle werden durch *Ströme* – endliche oder unendliche Sequenzen, die alle auf dem Kanal verschickten Nachrichten enthalten – modelliert, die Komponenten als (mengenwertige) Funktionen zwischen Ein- und Ausgabekanalgeschichten. Als Beschreibungstechnik für Komponenten werden oft Zustandsmaschinen verwendet. Die genaue Zuordnung von Komponentensemantik zu Zustandsmaschinen ist in [5, 6] dargestellt.

Solche Zustandsmaschinen sind erweiterte Mealy-Automaten deren Transitionen beschriftet sind mit logischen Bedingungen über die Datenwerte an den Eingangskanälen und mit Ausgabeanweisungen zur Steuerung der verbundenen Komponenten. Zusätzlich können sie Attribute besitzen, deren Werte in den Transitionen ebenfalls verwendet werden können. In den grafischen Spezifikationen lautet eine vollständige Beschriftung der Transitionen wie folgt:

$$pre;\ input\ /\ output;\ post$$

Angenommen, x bezeichnet das Eingabetupel und y das Ausgabetupel. s und s' seien der Systemzustand (die Werte der Attribute) vor und nach der Transition. Dann gilt:

- pre ist ein Prädikat über s, welches die Vorbedingung einer Transition beschreibt,
- $input$ ist ein Prädikat über s, x das die erforderliche Eingabe beschreibt. Negation ($\neg$) spezifiziert hier die Abwesenheit eines Signals.
- $output$ definiert mit Hilfe von s, x, y die Ausgabe der Transition,
- $post$ ist ein Prädikat über s, x, y, s' zur Definition der Zustandsänderung des Attributs.

In unserem Modell beeinhalten alle Zustände eine implizite Transition, deren Schaltbedingung die Negation aller anderen Transitionen ist. Sie wird ausgeführt, wenn zum Zeitpunkt eines Systemtakts keine andere Transition möglich ist. Damit wird die intuitiv naheliegende Reaktion auf nicht erkannte Eingaben gewährleistet, nämlich das Verharren im gegenwärtigen Zustand, ohne daß Ausgabe produziert wird.

Diese Leerschritte sind nicht die einzige Möglichkeit, unspezifizierte Transitionen zu interpretieren. So könnte z.B. das Verhalten eines Automaten völlig chaotisch werden, sobald

keine Transitionsbedingung mehr erfüllt ist. Diese Interpretation führt zu einem leicht handhabbaren mathematischen Verfeinerungsbegriff zwischen Spezifikationen und eignet sich gut für Requirements Engineering, bei dem das Verhalten eines Systems in mehreren Schritten nach und nach detaillierter spezifiziert wird.

Die Komponenten der Liftsteuerung arbeiten synchron: in jedem systemweiten Taktschritt führt jede Zustandsmaschine genau eine Transition aus. Diese Annahme ermöglicht Spezifikationen ohne Modellierung von Nachrichtenpuffern. Bei reaktiven Systemen – die oft mit wenig Speicher auskommen müssen – sind solche Puffer problematisch, da aus den Komponentenspezifikationen nicht offensichtlich hervorgeht, wie groß sie zu wählen sind.

Spezifikationen (auch grafische) großer Systeme werden ohne geeignete Hierarchiekonzepte schnell unübersichtlich. Die FOCUS-Methodik erlaubt an mehreren Stellen, Hierarchie einzuführen. So können Komponenten durch interne Komponentennetzwerke realisiert sein oder Zustände weitere Zustandsdiagramme enthalten. Für Komponentendiagramme ergibt sich die Semantik dieser Hierarchie direkt aus dem FOCUS-Systemmodell [7]; die Erweiterung flacher zu hierarchischen Zustandsdiagrammen wird in [8] behandelt.

3 Das Aufzugmodell

Das Modell selbst besteht aus Fischertechnik-Bausteinen und ist über eine Schnittstellenplatine (mit den nötigen Relais und Leistungstransistoren) an einen Microcontroller angeschlossen. Jedes Stockwerk besitzt einen Sensor für die Kabine, einen Rufknopf und eine Lampe, die den Ruf anzeigt. Die Tür der Kabine ist mit zwei Sensoren ausgestattet. Als Mikrocontroller dient ein BASIC-Tiger.

Von einem Aufzug werden neben einigen elementaren Sicherheitsanforderungen vor allem Effizienz und Fairness erwartet. Fairness bedeutet hier, daß jede Anforderung in endlicher Zeit bedient werden muß; und dies natürlich mit maximaler Effizienz. Somit ist die einfache Bedienstrategie, die sich an der zeitlichen Reihenfolge der Anforderungen orientiert, ausgeschlossen: sie ignoriert auf dem Weg liegende Anforderungen. Stattdessen bewegt sich die Kabine, wenn sie einmal in Bewegung ist, solange in dieselbe Richtung wie Aufträge für diese Richtung vorliegen. Dies ist die für Einkabinenaufzüge übliche Bedienstrategie [2].

Abbildung 1 zeigt die Systemstruktur des Aufzugs. Die Steuerung ist modular konzipiert und besteht aus folgenden Komponenten: Die Zentralkomponente läßt die Kabine nach der beschriebenen Bedienstrategie nach oben oder unten fahren. Die Tür des Aufzugs wird von einer eigenen Komponente gesteuert, die neben dem Öffnen und Schließen der Tür auch eine Sicherheitslichtschranke überwacht. Für jedes der vier Stockwerke steuert eine eigene Komponente die Anforderungsknöpfe und -lampen. An den Kanälen stehen die Nachrichten, die über die Kanäle fließen können.

Jede dieser sechs Komponenten wird zunächst mit einer eigenen Zustandsmaschine spezifiziert. Da der Status des Aufzugs in den Knoten der Zustandsmaschinen enthalten ist, kommen diese ohne zusätzliche Attribute aus. Aus diesem Grund sind die Vor- und Nachbedingungen immer $true$ und werden deswegen weggelassen. Auch die Angabe des Kanals, auf dem ein Datum empfangen oder gesendet wird, ist hier überflüssig, da die Nachrichtenmengen der Kanäle disjunkt sind.

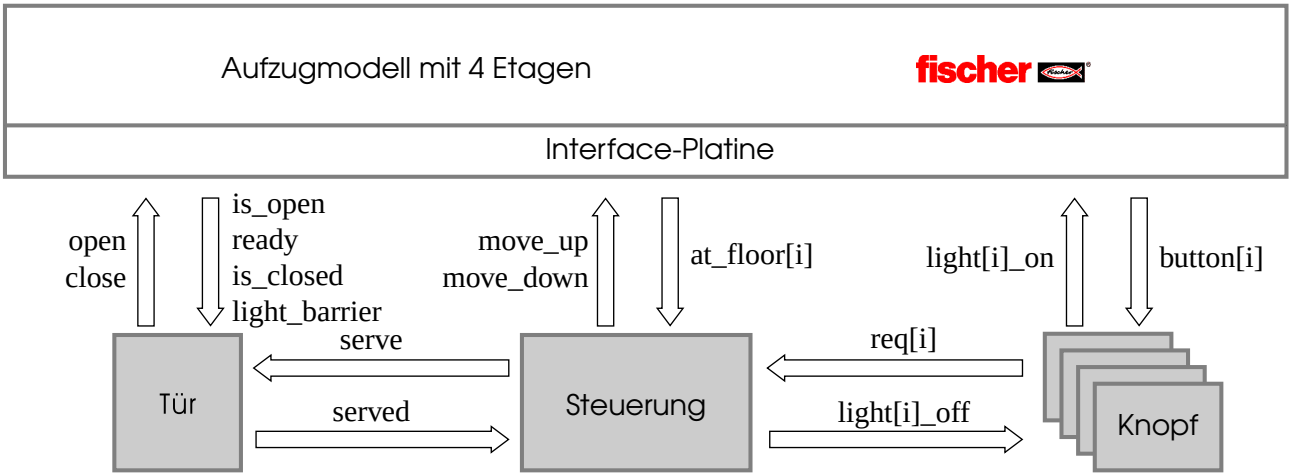

Abbildung 1: Systemstrukturdiagramm des Aufzugs

Das Verhalten der Zentralkomponente ist in Abbildung 2 zu sehen: Nach der Initialisierung des Systems befindet sich der Lift im untersten Stockwerk. Von dort aus werden, abhängig von den Anforderungen, die anderen Stockwerke bedient.

Die Beschriftung der Transitionen ist in Abbildung 3 am Beispiel der Zustände „III up“ und „III down“ dargestellt. Die übrigen Transitionen sind analog beschriftet. An den Transitionen zwischen den Up-Zuständen und den Down-Zuständen ist die Bedienstrategie gut zu sehen: Die Kabine wechselt erst dann die Richtung, wenn explizit keine Anforderung derselben Richtung vorliegt: $(\neg req_3 \wedge \neg req_4)$, bzw. $(\neg req_1 \wedge \neg req_2)$.

Die Türkomponente Abbildung 4 beginnt auf das Signal *serve* mit dem Öffnen- und Schließzyklus. Sollte die Lichtschranke während des Schließzyklus unterbrochen werden, wird die Tür sofort wieder geöffnet. Auf das Signal *served* nimmt die zentrale Steuerung die Bearbeitung weiterer Aufträge wieder auf.

Solange ein Knopf im Zustand *on* ist (Abbildung 5) und keine Nachricht *light-off* anliegt, sendet der Knopf wiederholt eine Anforderung pro Systemtakt an die Zentralkomponente. In diesem synchronen Modell ohne Pufferung der Nachrichten wird auf diese Weise ein permanentes Signal simuliert.

4 Code-Erzeugung

Im nächsten Entwicklungsschritt werden die Zustandsmaschinen in die synchrone Programmiesprache ESTEREL [3] übersetzt. ESTEREL ist eine imperative synchrone Programmiersprache, die besonders auf reaktive Systeme mit komplexen Kontrollstrukturen zugeschnitten ist; für Signalverarbeitung etwa ist sie weniger geeignet. Das Berechnungsmodell ist gerade die zyklische Reaktion auf Eingabesignale, wie sie den von uns verwendeten Mealy-Automaten zugrunde liegt.

Die Übersetzung der Zustandsautomaten in ESTEREL folgt dem in [1] vorgestellten Schema. Abbildung 6 zeigt einen Ausschnitt des Übersetzungsergebnisses.

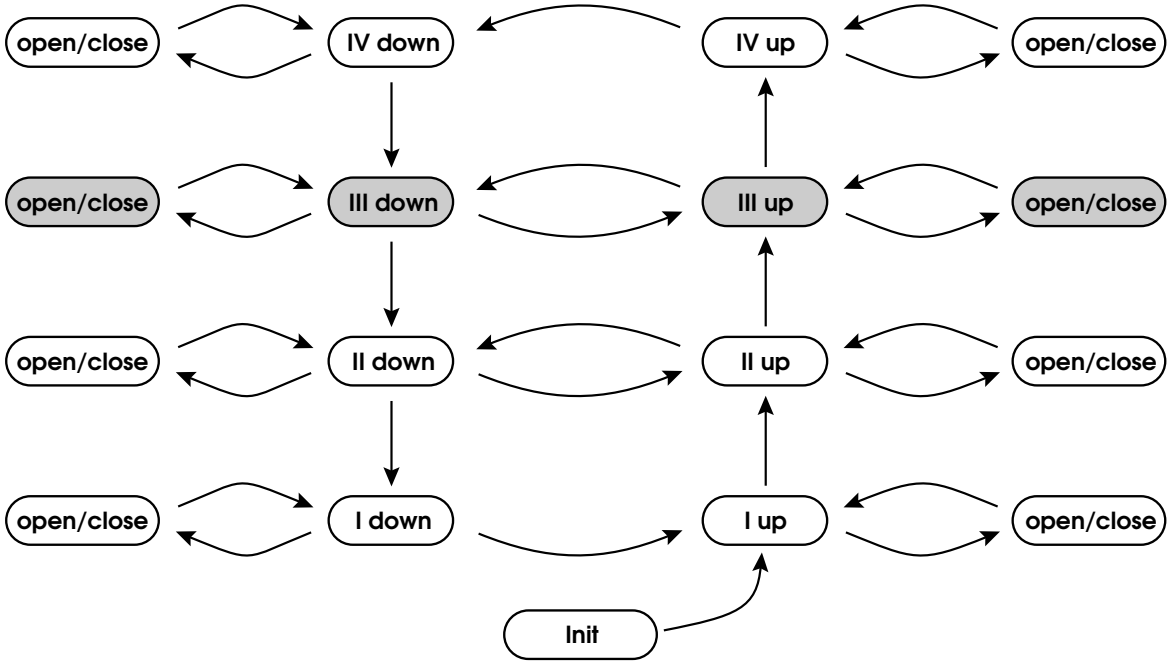

Abbildung 2: Zustandsmaschine der zentralen Steuerung

Der ESTEREL-Compiler übersetzt den ESTEREL-Code der Zustandsmaschinen in portable und effiziente C-Programme.

Der generierte C-Code ist sehr einfach, wenn auch nicht besonders übersichtlich: er besteht im wesentlichen aus drei Teilen: der Deklaration von Zustandsvariablen und Ein-/Ausgabekanälen (im synchronen Modell reichen dazu einfache Variablen), einer Initialisierungsfunktion, und der eigentlichen Reaktionsfunktion, die aus einer Folge von Zuweisungen an boolesche Variablen besteht. Er kann deshalb mit einem simplen PERL-Skript (etwa 500 Zeilen) in den vom BASIC-Tiger verwendeten, etwas eingeschränkten BASIC-Dialekt übersetzt werden.

ESTEREL-Programme lassen sich als synchrone FOCUS-Komponenten interpretieren [5], und mit den für FOCUS entwickelten Techniken zu Datenflußnetzwerken zusammenfassen. Auf der Programmebene erfolgt diese Integration (die der Struktur von Abbildung 1 entspricht) noch von Hand: dazu müssen die Aus- und Eingänge der Komponenten verbunden werden.

Das so entstandene Tiger-BASIC-Programm kann nun nach einigen hardwarebedingten Anpassungen (wie z. B. der Zuweisung der Steuersignale zu I/O-Ports) in Maschinencode übersetzt und in den E^2PROM des BASIC-Tigers übertragen werden.

5 Verifikation

Ein Entwurfsprozeß auf der Basis von Beschreibungstechniken mit präziser Semantik erlaubt die Verifikation von Systemeigenschaften mit mathematischen Techniken. Bei kleinen bis mittleren Systemen eignet sich dazu *Model Checking*, bei dem der gesamte Zustandsraum

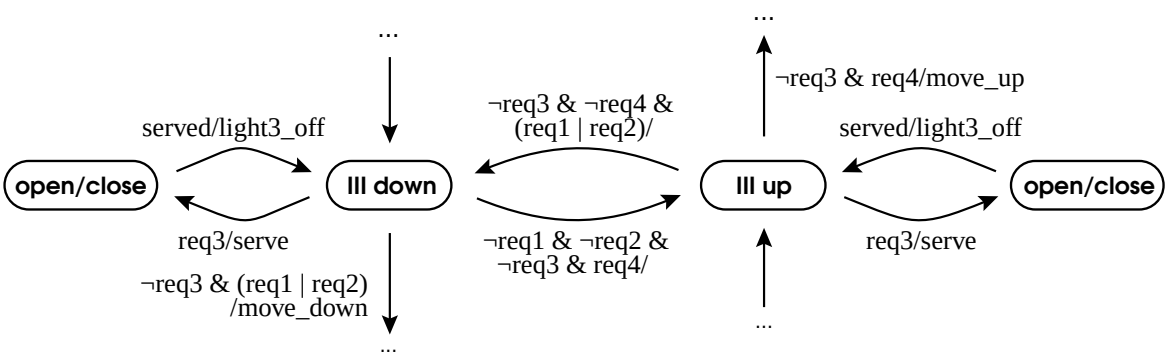

Abbildung 3: Ausschnitt aus der zentralen Steuerung

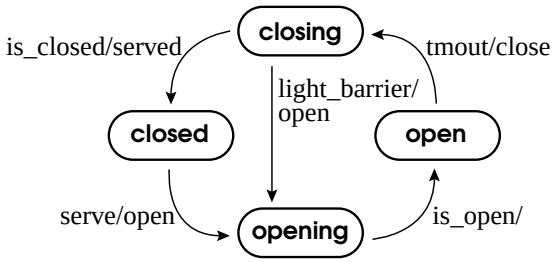

Abbildung 4: Zustandsmaschine der Türsteuerung

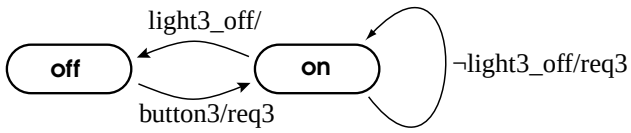

Abbildung 5: Eine der vier Knopfsteuerungen

```
case s3Up do
   await
     case Req3 do
        emit Serve; await Served; emit s3Up

     case [Req4] do
        run GoUp[signal At4 / target]; emit s4Up

     case [Req1 or Req2] do
        emit s3Dn
   end await

case s3Dn do
   await
     case Req3 do
        emit Serve; await Served; emit s3Dn

     case [Req1 or Req2] do
        run GoDn[signal At2 / target]; emit s2Dn

     case [Req4] do
        emit s3Up
   end await
```

Abbildung 6: Auszug aus dem ESTEREL-Programm

eines Systems mit hocheffizienten Algorithmen durchsucht wird. Systemeigenschaften werden dabei meist mit temporalen Logiken beschrieben. Verbreitete Logiken dieser Art sind die *Computation Tree Logic* CTL, und die *lineare temporale Logik* LTL. Die bekanntesten Werkzeuge für Model Checking sind SMV [11] für CTL, SPIN [9] für LTL und der Model Checker des STeP-Projekts [4], der ebenfalls LTL-Formeln überprüft.

Der beschriebene Entwurfsprozeß bietet mehrere Schnittstellen zu solchen Verifikationswerkzeugen (Abbildung 7).

- Auf der Ebene der Zustandsmaschinen können, je nach zugrundeliegender Logik, Model Checker wie SMV oder STeP eingesetzt werden. Im allgemeinen ist Verifikation auf dieser Ebene am effizientesten, da die Spezifikationen noch nicht mit den Implementierungsdetails von ESTEREL oder der Zielsprache angereichert und somit am abstraktesten sind.
- Für ESTEREL existieren spezielle Werkzeuge, die Eigenschaften auf der Basis einer LTL-artigen Logik überprüfen können. Mit diesen Werkzeugen können allerdings nur einzelne Komponenten verfiziert werden.
- Schließlich lassen sich die vom ESTEREL-Compiler erzeugten Programme nicht nur in BASIC, sondern auch in die Eingabesprache PROMELA des Model Checkers SPIN

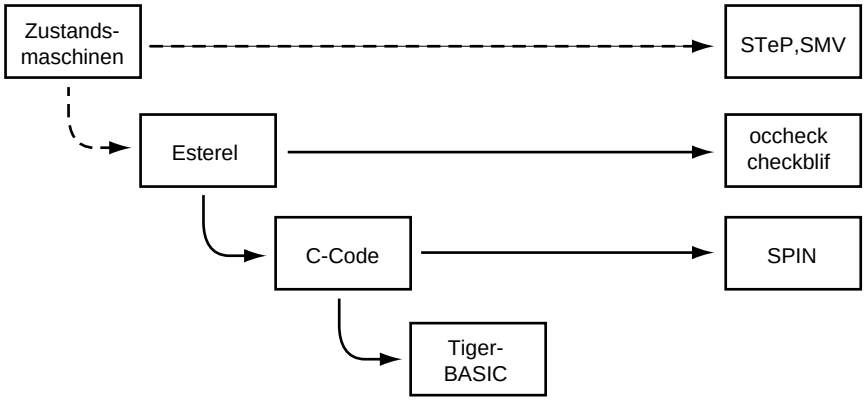

Abbildung 7: Entwurfschritte und Verifikationswerkzeuge

übersetzen. Auf dieser Ebene können auch Netzwerke von Komponenten überprüft werden.

In Abbildung 7 kennzeichnen die gestrichelten Linien die Übersetzungsschritte, die gegenwärtig noch von Hand durchgeführt werden.

Elementare Sicherheitseigenschaften, etwa daß der Aufzug sich nur mit geschlossenen Türen bewegt, lassen sich mit jedem dieser Werkzeuge zeigen. Interessanter ist die Forderung, daß jede Kabinenanforderung vom Aufzugsystem beantwortet werden soll. Diese Eigenschaft gilt in unserem Modell natürlich nur, wenn sichergestellt ist, daß der Lift nicht unendlich lange durch die Sicherheitslichtschranke oder den Rufknopf blockiert wird. Abbildung 8 zeigt die Formalisierung dieser Eigenschaft in LTL. Rechts von dem Implikationszeichen steht die eigentliche Forderung: nach jeder Anforderung (req_i) soll der Aufzug das Stockwerk besuchen ($floor = i$). Die Konjunktion links davon beschreibt die Voraussetzung, unter der diese Eigenschaft gilt: befindet sich die Kabine in einem Stockwerk, wird sie es auch irgendwann verlassen, bevor das nächste Mal dieses Stockwerk angefordert oder die Lichtschranke unterbrochen wird.

```
(   [](floor=1 --> <>((req[1] \/ light_barrier) Awaits ~(floor=1)))
 /\ [](floor=2 --> <>((req[2] \/ light_barrier) Awaits ~(floor=2)))
 /\ [](floor=3 --> <>((req[3] \/ light_barrier) Awaits ~(floor=3)))
 /\ [](floor=4 --> <>((req[4] \/ light_barrier) Awaits ~(floor=4))))
        -->  [](req[i] --> <>(floor=i))
```

Abbildung 8: Fairness des Aufzugs

6 Zusammenfassung

Das Beispiel zeigt, daß sich ein formal fundierter Entwurfsprozeß weitgehend automatisieren läßt. Dazu werden mehrere orthogonale Arbeiten aus der Informatik verbunden:

- Synchrone Programmiersprachen wie ESTEREL, die sich durch ihre präzise Semantik und effiziente Code-Erzeugung auszeichnen, und sich gut für zyklisch arbeitende reaktive Systeme eignen;
- die Systemmodelle von FOCUS, mit denen die Komposition synchroner Programme definiert werden kann;
- effiziente automatische Verifikationstechniken, die zur Überprüfung wesentlicher Systemeigenschaften verwendet werden.

Im gezeigten Entwicklungsprozeß sind die Entwurfsschritte noch nicht vollständig integriert. So werden die Zustandsmaschinen dieses Beispiels zwar schematisch, aber noch von Hand in ESTEREL übertragen. Die anschließenden Codetransformationen sind zwar automatisiert, die entstehenden Programme müssen aber noch von Hand komponiert werden.

Das an der TU München entwickelte CASE-Tool AUTOFOCUS [10] für verteilte und reaktive Systeme wird gegenwärtig erweitert, um u.a. auch die hier vorgeschlagene Code-Erzeugung aus Zustandsdiagrammen integriert und automatisch durchführen zu können. Damit lassen sich dann auch größere Systeme spezifizieren und nahezu vollständig automatisiert effizienter Code generieren. Lediglich für die Schnittstelle zu den I/O-Ports des Mikrocontrollers muß noch von Hand programmierter Code entwickelt werden; da diese Schnittstelle von Anwendung zu Anwendung stark variiert, bietet sie weniger Möglichkeiten zur Automatisierung.

Seit einiger Zeit gibt es insbesondere für die Automobilelektronik Bemühungen, die Steuergeräte eingebetter Systeme zu vernetzen. An der TU München laufen deshalb Arbeiten, einfache reaktive Modelle, wie sie für den Aufzug verwendet wurden, mit asynchron kommunizierenden Ansätzen zu verbinden.

Literatur

[1] C. André. Represenation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96*, July 1996.

[2] G.C. Barney and S.M. Dos Santos. *Lift Traffic Analysis, Design and Control*. Peter Peregrinus, 1977.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[4] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-Algorithmic Verification of Reactive and Real-time Systems. In *CAV'96*, LNCS 1102, pages 415–418, 1996.

[5] M. Broy. Abstract Semantics of Synchronous Languages: The Example Esterel. Technical Report TUM-I9706, 1997.

[6] M. Broy. The Specification of System Components by State Transition Diagrams. Technical Report TUM-I9729, 1997.

[7] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems: An Introduction to Focus—Revised Version. Technical Report TUM-I9202-2, 1993.

[8] R. Grosu, G. Ştefănescu, and M. Broy. Visual formalisms revisited. In L. Lavagno and W. Reisig, editors, *International Conference on Application of Concurrency to System Design*. IEEE Computer Society Press, 1998.

[9] G. J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[10] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus—a tool for distributed systems specification. In *Proceedings FTRTFT'96 — Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 1135, 1996.

[11] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.